Document convergence in an interactive formatting system

by Donald D. Chamberlin

One of the most complex aspects of document formatting is the processing of references to remote objects such as headings and figures. In the case of a forward reference to an object that occurs later in the document, two formatting passes are usually needed before the document converges to a stable state. Some documents require more than two passes to converge, and cases are known of documents that never converge but oscillate between two unstable states. This paper describes the techniques used for resolving references and detecting document convergence by the Interactive Composition and Editing Facility, Version 2 (ICEF2). ICEF2 is an interactive formatting system that allows users to move about in a document, editing and reformatting pages. The concepts of *formatting pass* and *document convergence* are discussed in the context of interactive formatting. A description is given of the ICEF2 data store, a small relational database manager with special features for detecting document convergence. A sample ICEF2 style definition is discussed to illustrate how ICEF2 deals with document elements whose appearance depends on their location on the page.

Introduction

A document formatter is a computer program that helps users to prepare documents for printing by performing tasks that are tedious to do manually, such as hyphenation and justification of text. Document formatters can be classified by various criteria [1], including their degree of interactivity and the types of formatting functions they provide.

Most formatting systems designed before 1980 process an input or "source" document, containing unformatted text and "markup," and generate a formatted output document, usually without interacting with a human user. The markup contained in the source document may consist of low-level commands that direct the system to use a specific font, start a new page, etc. Alternatively, the markup may consist of higher-level "tags" that label objects such as chapter headings, footnotes, examples, and lists, leaving the detailed appearance of these objects up to a "style definition" that is external to the document. Systems that accept descriptive tags and process them according to an external style definition are called "generic markup" systems. Since generic markup deals with the logical structure of the document rather than with its physical appearance, systems of this type are able to provide many advantages such as device independence and enforcement of style standards. Well-known examples of generic markup-type systems include IBM's Generalized Markup Language [2] and Brian Reid's Scribe® [3].

During the processing of a document, a formatting system may encounter a command or tag that cannot be fully processed with the data currently available—for example, a reference to a figure that has not yet been encountered. Most systems respond to this situation by making some reasonable assumption about the missing information—perhaps

replacing it with a symbolic name—and continuing to format the remainder of the document. When the system reaches the end of the document, additional formatting passes may be called for if the document has not yet been completely formatted. Each pass makes use of the information gathered about the contents of the document during previous passes. We say that a document has "converged" at the end of a pass if additional passes (in the absence of user editing) would not result in any changes to the content or format of the document.

Markup-type systems often provide facilities for preserving information from one pass to the next, and for controlling the number of passes. Typical of these facilities are the @Label, @Ref, and @PageRef commands of Scribe [4]. An @Label command may be used anywhere in a document to associate a symbolic name with the current section number and page number. The user can then cause the saved section number and page number to be inserted at any other place in the document by invoking the @Ref and @PageRef commands with the correct symbolic name. An @Ref or @PageRef command that is encountered before its matching @Label command is called a "forward reference." When a document is formatted for the first time, forward references are resolved by printing the symbolic name in place of the missing section or page number. When the end of the document is reached, all the section and page numbers accumulated during the formatting run are saved, with their symbolic names, in an "auxiliary" file keyed to the name of the document. On later formatting runs, the information in the auxiliary file is used to resolve forward references. Scribe performs one formatting pass each time it is invoked, and leaves it up to the user to determine whether the document has converged or whether additional passes are needed.

Similar facilities for saving information in an external file during the processing of a document are provided by Donald Knuth's TEX™ system [5] and by IBM's Document Composition Facility (DCF) [2]. In TEX, commands named \write and \read are used in macros for storing computed data in an auxiliary file and retrieving them later. In DCF, an option called "SYSVAR W" causes the system to write all data needed to resolve references into an auxiliary file, and a "SYSVAR R" option causes the system to read these data from a named file at the beginning of a session. DCF also provides a user-specifiable option called "TWOPASS" that forces the formatter to make two formatting passes over the document, preserving reference information from one pass to the next. However, neither TEX nor DCF provides any built-in facilities for determining whether a document has converged at the end of a pass.

In the last few years, the emergence of intelligent workstations and fast, high-resolution graphic displays has made it possible for document-formatting systems to become much more interactive. Interactive formatters usually provide one or both of the following properties:

1. Systems that provide the user with a formatted display that accurately represents fonts, line breaks, and pagination are usually called "What You See Is What You Get," or "WYSIWYG," systems. A true WYSIWYG system supports an interactive interface that allows the user to reformat and examine portions of the document on the display without reformatting the entire document. By giving the user a quick way to see the effects of a change in its local context, WYSIWYG systems have the potential to improve user productivity and save much of the paper and computing resources that would be used in repeated reformatting of the entire document.
2. Ben Shneiderman has coined the term "direct manipulation" [6] to describe text-editing systems that allow users to perform insertions, deletions, and other operations directly on the formatted page rather than on a separate "markup" representation of the document. In a direct-manipulation system, each user-initiated action takes effect immediately on the formatted page and provides direct visual feedback. By allowing the user to interact with the document itself rather than with an abstract command language, direct-manipulation systems have the potential to reduce learning time and to make formatting systems more accessible to unsophisticated users.

The concepts of WYSIWYG and direct manipulation were pioneered by the Alto personal computer at the Xerox Palo Alto Research Center [7] and by the Etude system at MIT [8]. Commercial systems based on these ideas include Apple's MacWrite™ [9], the Xerox Star [10], Interleaf™, Texet®, Xyvision®, and others [11]. Related research projects in academic environments include the Lara editor at ETH (Swiss Federal Institute of Technology) [12] and the Andrew system at Carnegie-Mellon University [13].

While many systems provide both the WYSIWYG and direct-manipulation properties, it is not necessary that these properties always go together. For example, a word processor might support a direct-manipulation interface on a personal computer display using simple monospaced fonts and fixed line spacing; the document might then be reformatted using typographic fonts for printing. Such a direct-manipulation system would lack the WYSIWYG property because the display could not be used to make decisions about line and page breaks in the printed document. Conversely, a system could present the user with an accurate display of the document to be printed, but could require editing changes to be made on a markup representation of the document rather than directly on the formatted page. Such a system might be considered a WYSIWYG system without a direct-manipulation feature.

In a direct-manipulation system, the document being edited *always* exists in a formatted state. Rather than starting at the beginning of the document and proceeding to the end,

the system must be able to apply editing changes to the middle of a formatted document, incrementally reformatting the local area where the change occurred. In principle, a small editing change might have widespread and complex effects on a document. For example, deleting a figure might affect other figure numbers, the list of illustrations, and many figure references throughout the document. However, most existing direct-manipulation systems have a relatively simple document model in which editing changes simply ripple forward until they reach a natural boundary such as the end of a paragraph or section. Developing direct-manipulation techniques based on more complex document models raises interesting questions of document convergence and is a fruitful area for research.

Document convergence in ICEF2

IBM's Interactive Composition and Editing Facility, Version 2 (ICEF2) [14, 15] is an attempt to combine the advantages of a WYSIWYG display with the device independence and complex processing provided by generic markup. ICEF2 is based on an experimental formatting system named Janus, which was described in an earlier paper [16]. ICEF2 is a generic-markup system that employs the Generalized Markup Language (GML) for marking up source documents. The GML "tags" in the source document are interpreted according to an external style definition consisting of "tag routines" written in a high-level structured programming language [17]. However, unlike most generic-markup systems, ICEF2 does not format an entire document without user intervention. Instead, ICEF2 presents two views of the document to the user: the source view, consisting of unformatted text and GML tags, and the formatted view, consisting of pages of formatted text in typographic fonts, just as they will appear when printed. Depending on the type of terminal in use, the user can see these two views simultaneously on two displays, or can switch from one view to the other on a single display. The user can page forward or backward in either view, and the system automatically moves the other view to a consistent position in the document.

By displaying formatted pages to the user, ICEF2 provides the direct visual feedback of a WYSIWYG system. However, when the user wishes to make editing changes to the document, the changes are not applied directly to the formatted pages, but to the markup from which these pages are derived. The system then reformats the pages that were affected by the changes. The user can apply a series of editing changes to the markup view and then call for reformatting of the affected pages by invoking the command SHOW. The system automatically determines which pages require reformatting. One page is the smallest unit of reformatting supported by ICEF2. Because the ICEF2 user applies editing operations to the markup view rather than to the formatted view, ICEF2 is not a "direct-manipulation" system. However, ICEF2 is much more interactive than typical generic-markup systems because it allows a user to move about in the document, applying editing changes and reformatting pages incrementally.

In order to support single-page reformatting, ICEF2 saves a "stub" file in secondary storage just before beginning the formatting of each page. The stub contains the complete formatting state at the beginning of the page. ICEF2 also saves each complete formatted page in secondary storage. The stub files permit the system to resume formatting at the beginning of any page in the document that it has previously visited. When editing changes are applied to a page, the formatter reformats the page, beginning with its stub. In general, reformatting a page may cause text to ripple forward, affecting the contents of the following pages. Therefore, ICEF2 maintains a "Safe Mark," which is defined as the most advanced stub in the document that is still valid—i.e., has not been affected by editing changes. Any change applied to the text or markup of a document moves the Safe Mark back to the page on which the change occurred (unless the Safe Mark is already on an earlier page). If the user asks to see a formatted page that precedes the Safe Mark, it can be displayed immediately without reformatting. If the user asks to see a formatted page that falls after the Safe Mark, ICEF2 begins at the Safe Mark and formats forward to the desired page. During this process, new stubs are created and the Safe Mark is advanced.

When ICEF2 encounters a GML tag that cannot be fully processed with currently available information—for example, a forward reference to a figure—it simply substitutes a *?* for the unavailable data. For example, a figure reference may appear as *Fig. ? on page ?*. During the course of editing and formatting, the missing objects will be encountered and the question marks will gradually disappear. When the user wishes to ensure that the document is in final form, he may use the PERFECT command, which forces the system to do as much work as necessary to resolve all references, propagate all ripple effects, and prepare the document for final printing. After the PERFECT command, the Safe Mark is at the end of the document, and any page may be viewed in its final form without reformatting. Of course, if editing changes are applied after a PERFECT command, the Safe Mark moves back to the edited page, and another PERFECT command is needed before the document is ready for printing.

Like other markup-type systems, ICEF2 saves information accumulated about a document during formatting in an external file called the "data store," which persists from one ICEF2 session to another. Each document has its own data store. By using information in the data store, ICEF2 may be able to resolve a reference or generate a table of contents based on information accumulated in a previous session. Of course, if the document is edited, the information in the data store may become obsolete and need to be updated.

The Reference Tag:

```
:figref id='bicycle'.
```

The Antecedent Tag:

```
:fig id='bicycle'.
```

Example 1

Syntax of GML tags.

In an interactive system such as ICEF2, the concept of a "pass" is not straightforward, since the user and the system alternate in editing and reformatting portions of the document. During a session, the Safe Mark may advance and retreat many times as changes are applied to various pages. In the ICEF2 environment, we define a "pass" as any sequence of editing and formatting actions that advances the Safe Mark from the beginning of the document to the end of the document. An ICEF2 pass may involve repeated editing and reformatting of various portions of the document. As a result, individual items of information in the data store, such as the number of the page on which a particular figure is placed, may be updated multiple times during a pass. **Figure 1** illustrates the difference between the concept of a "pass" in a batch-type system and the equivalent concept in ICEF2.

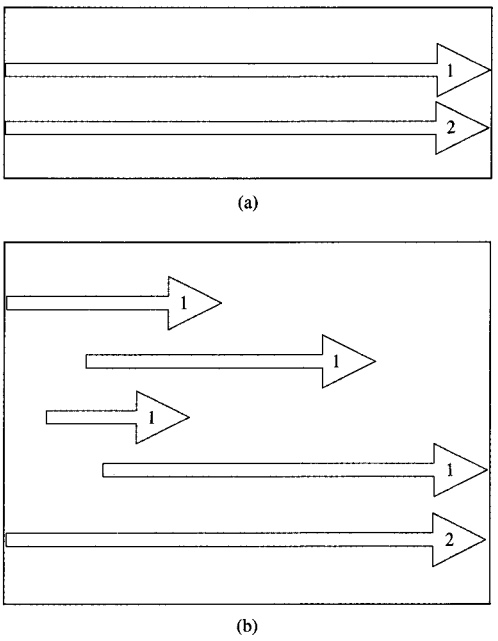


Figure 1

How the concept of a "pass" applies to ICEF2: (a) Passes in a batch-type formatter; (b) passes in ICEF2.

In the ICEF2 environment, it would be inappropriate to expect the user to specify in advance how many passes are required to format a document, since this number depends on the editing actions that occur during the ICEF2 session. Therefore, ICEF2 automatically determines the number of passes that are required. When the user invokes the PERFECT command, the system begins formatting at the Safe Mark and completes the current pass. It then examines the data store for the document to determine whether the document has converged. The ICEF2 data store has some special features designed to detect convergence of the document. If, at the end of a pass, the document has not yet converged, ICEF2 performs additional passes, up to a user-controlled limit, until convergence is achieved. These system-initiated passes always start at the beginning of the document and are performed without user intervention.

The remainder of this paper examines the mechanisms used in the ICEF2 data store to detect convergence and considers some specific cases of documents and style definitions that interact to provoke multiple passes.

Multiple-pass documents

The need for more than one formatting pass is typically caused by *references* in the document. A reference is a tag that requires, for its processing, data supplied by another tag. We refer to the tag that supplies the missing data as the *antecedent* of the reference. If the reference occurs in the document after its antecedent, we call it a *backward reference*; if the reference occurs before its antecedent, we call it a *forward reference*. A reference tag and its antecedent are typically linked together by matching id attributes, as in **Example 1**.

If the antecedent tag is encountered first in the document, its processing generates and stores the data needed for processing of the reference tag: for example, the facts that the figure with id='bicycle' is Figure 5 and is located on page 21. When the reference tag is encountered, these data are used to resolve the reference into a character string such as *Figure 5 on page 21*.

In the case of a forward reference, however, the reference is encountered before the antecedent. In such a case, ICEF2 looks for the figure-placement data associated with id='bicycle' and, failing to find them, substitutes a question mark for each missing data item, resulting in the following resolution for the reference tag: *Figure ? on page ?*. The fact that data are missing does not, in itself, trigger an additional pass through the document, since the antecedent may simply be missing due to an error. If the antecedent tag is encountered later in the document, it stores the figure number and page number for id='bicycle' in the data

First Pass:
Reference resolves to *Fig. ? on page ?.*

Second Pass:
Reference resolves to *Fig. 27 on page 153.*
Growth of the reference string ripples forward, causing the antecedent figure to move to page 154.

Third Pass:
Reference resolves to *Fig. 27 on page 154.*
Antecedent remains on page 154.
Document has converged.

Example 2

A three-pass document.

store. A second pass is then required because a data item was *missing and later inserted* in the database during the current pass.

A similar case may be caused by a user who is scanning through a document, examining pages and editing them. When a forward reference tag is encountered, the data for its resolution may be available from previous passes through the document, and the reference may be resolved to the string *Figure 5 on page 21.* Subsequently, the user may apply editing changes (e.g., inserting additional figures) that cause the antecedent to be renumbered as Figure 7 and moved to page 35. When the antecedent tag is encountered, it updates its placement data in the data store. In this case, an additional pass is triggered by the fact that data are *read and later changed.*

Another case calling for multiple passes can be caused by deletion of material from a document during the editing and formatting process. Suppose that a "Table of Contents" tag, near the beginning of the document, examines the data stored by all the "Heading" tags during previous passes and generates a table of contents listing all the headings in the document. Then suppose that the user, during processing of the document, deletes a block of material containing one or more headings. The deletion of the heading tags does not in itself affect the data store; indeed, since the missing tags will never be encountered, the data associated with these tags will remain unchanged until the end of the pass. Nevertheless, another formatting pass is necessary to recompute the Table of Contents and remove the now-deleted headings; the need for another pass is triggered by the fact that certain data items were *read and never written* in the current pass. This case requires some special handling, in that some mechanism is needed to remove the *never-written* items from the data store before the beginning of the next formatting pass.

In summary, there are two types of tags that interact with the data store during formatting of a document: *reference tags*, which read data out of the store, and *antecedent tags*, which write data into the store. At the end of a pass through the document, an additional pass is called for if one of the following conditions has occurred during the pass just completed:

1. A data item is found to be missing and is later inserted.
2. A data item is read and later changed.
3. A data item is read but never written.

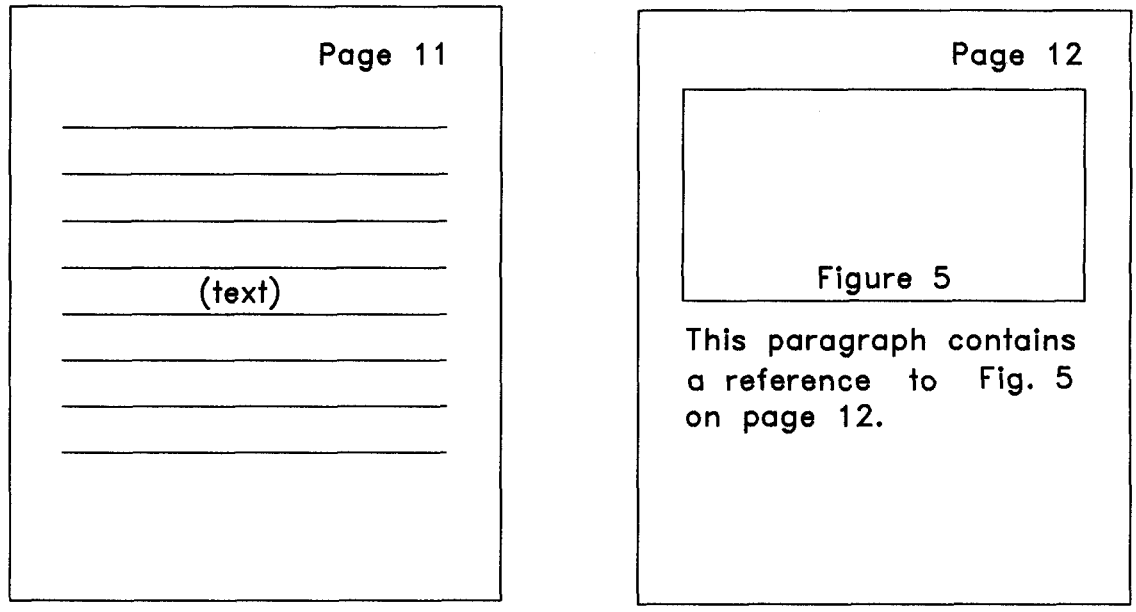
Nonconverging documents

It is obvious from the above discussion that two-pass documents are quite commonplace; in fact, any document containing a forward reference requires at least two passes to converge. Three passes are needed if the resolution of the reference during the second pass causes the reference string to grow longer, and this change ripples forward to affect the placement of the antecedent, as in **Example 2.**

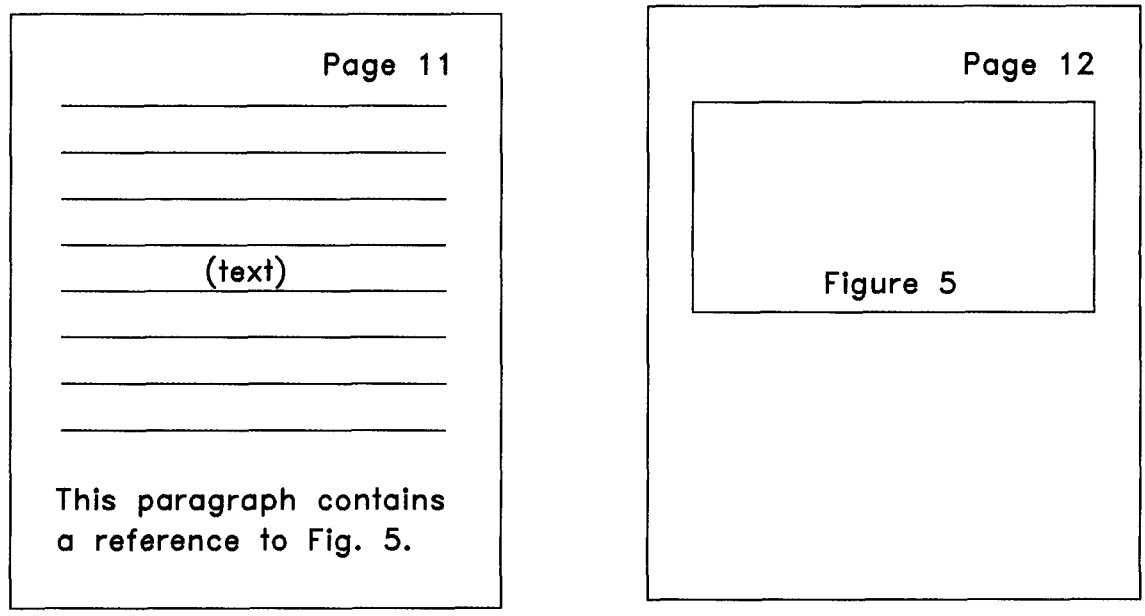
Since the program that processes a GML tag can perform any desired computation, it is easy to contrive an example of a document that never converges, no matter how many formatting passes are made. As such an example, consider a GML tag that simply counts passes, printing the number of each pass and storing it in the data store. A document containing such a tag would be a trivial example of a nonconverging document, since each formatting pass would read and then update the pass number in the data store.

Apart from contrived examples such as the one above, nonconverging documents are relatively rare, but at least one example is known of a "naturally occurring" document that fails to converge, oscillating instead between two unstable states on successive passes. The document uses the "Starter Set" of GML tags [2, 15]. The nonconverging property of the document is triggered by the following rules for processing the Starter Set tags:

1. The "paragraph" tag, :p., is processed in such a way that one line of a paragraph is never placed on a page by itself (unless the paragraph consists of only one line). This is called the "Widow Prevention Rule."
2. The "figure" tag, :fig., identifies a figure by an "id" attribute. Processing of this tag causes the figure number to be incremented and the figure to be placed at the top of the next available page after the occurrence of the tag. In order to be placed at the top of a page, the figure is allowed to "float" out of sequence with respect to the surrounding text. (Various other placement options are supported by this tag, but the option described here is the default and was used in the document of interest.)
3. The "figure reference" tag, :figref., has a "refid" attribute that matches the "id" attribute of the antecedent :fig. tag. The resolution of the :figref. tag proceeds as follows:
   a. If the figure reference is on the same page as its



(a)



(b)

Figure 2

Example of an oscillating document: (a) Document state after odd-numbered passes; (b) document state after even-numbered passes.

```
(... misc. text. ...)
:fig id='abc'.
(... contents of figure ...)
:efig.
:p.
This paragraph contains a reference to
:figref refid='abc'..
```

Example 3

Markup for document shown in Figure 2.

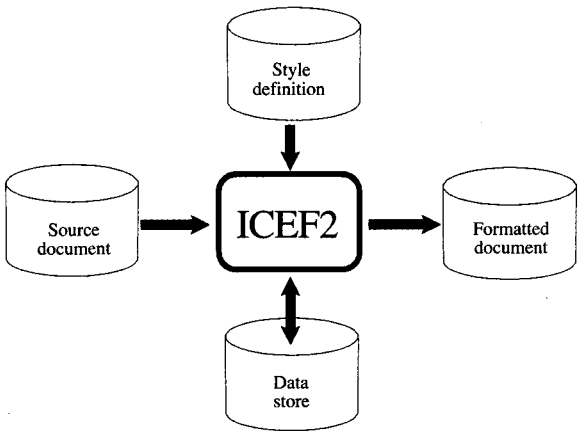


Figure 3

The formatting process in ICEF2.

antecedent figure, it resolves to *Figure X* (using the correct figure number for X).

b. If the figure reference is *not* on the same page as its antecedent figure, it resolves to *Figure X on page Y* (using the correct figure number and page number).

On the basis of the above rules, we can now explain the behavior of the oscillating document. The two unstable states of the document are illustrated in **Figure 2**. The document contains the markup shown in **Example 3**.

On the first formatting pass, the figure "floats" ahead of the figure reference, which is resolved to *Figure ? on page ?*. The figure is placed at the top of page 12. The figure reference occurs at the bottom of page 11 in a three-line paragraph; but there is not room for three lines at the bottom of page 11, and the Widow Prevention Rule forces the entire paragraph to page 12, just below the figure.

On the second (and all even-numbered) formatting passes, the :figref. tag routine fetches the figure number and page number of the antecedent from the data store, and notices that the reference is on the same page as the antecedent. Therefore the reference is shortened to *Figure 5*. This shortens the paragraph containing the reference to two lines and allows it to fit on page 11.

On the third (and all subsequent odd-numbered) formatting passes, the :figref. tag routine fetches the figure number and page number of the antecedent, and notices that the reference is *not* on the same page as the antecedent. Therefore the reference is expanded to *Figure 5 on page 12*. This lengthens the paragraph containing the reference to three lines, and the Widow Prevention Rule moves the three-line paragraph to page 12.

If ICEF2 is presented with an oscillating document such as the one in this example, it will perform passes up to a limit that can be controlled by the user (the default limit is two passes). The system will then inform the user that the document failed to converge after the given number of passes and will report the line number of the reference tag responsible for the nonconvergence.

The ICEF2 data store

The process by which ICEF2 formats a document is shown in **Figure 3**. The user can interact with the source document by using an editor (this interaction is not shown in the figure). When the user issues the SHOW or PERFECT command, ICEF2 reloads its internal state from the appropriate stub and begins formatting. It takes lines from the source file, beginning at the point corresponding to the selected stub. GML tags found in the source text are interpreted by "tag routines" in an external "style definition." Some sample style definitions are provided with ICEF2, and additional styles may be defined by the user. During the process of formatting the document, a tag routine can invoke certain commands that save information in the ICEF2 data store for use by later passes, or recall information saved previously.

The ICEF2 data store is maintained by a small, special-purpose relational database system called the Data Manager. The design of the Data Manager is based on the following assumptions:

- The amount of data to be stored for each document is small.
- All data items to be stored are of character-string type.
- Ordering of data is important (e.g., the chapter headings in the table of contents should appear in the proper order).
- Retrieval of data is done in simple ways (e.g., by simple key-matching).
- The Data Manager should incorporate the notion of a "pass" through the document and provide special mechanisms to detect changes that occur from one pass to another.

```
(... misc. text. ...)
:fig id='abc'.
(... contents of figure ...)
:efig.
:p.
This paragraph contains a reference to
:figref refid='abc'..
```

Example 3

Markup for document shown in Figure 2.

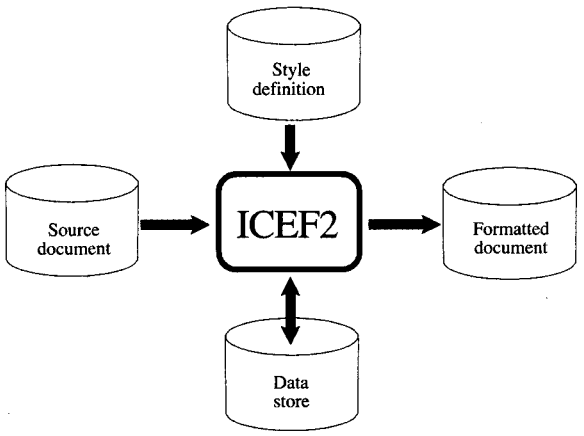


Figure 3

The formatting process in ICEF2.

antecedent figure, it resolves to *Figure X* (using the correct figure number for X).

b. If the figure reference is *not* on the same page as its antecedent figure, it resolves to *Figure X on page Y* (using the correct figure number and page number).

On the basis of the above rules, we can now explain the behavior of the oscillating document. The two unstable states of the document are illustrated in **Figure 2**. The document contains the markup shown in **Example 3**.

On the first formatting pass, the figure "floats" ahead of the figure reference, which is resolved to *Figure ? on page ?*. The figure is placed at the top of page 12. The figure reference occurs at the bottom of page 11 in a three-line paragraph; but there is not room for three lines at the bottom of page 11, and the Widow Prevention Rule forces the entire paragraph to page 12, just below the figure.

On the second (and all even-numbered) formatting passes, the :figref. tag routine fetches the figure number and page number of the antecedent from the data store, and notices that the reference is on the same page as the antecedent. Therefore the reference is shortened to *Figure 5*. This shortens the paragraph containing the reference to two lines and allows it to fit on page 11.

On the third (and all subsequent odd-numbered) formatting passes, the :figref. tag routine fetches the figure number and page number of the antecedent, and notices that the reference is *not* on the same page as the antecedent. Therefore the reference is expanded to *Figure 5 on page 12*. This lengthens the paragraph containing the reference to three lines, and the Widow Prevention Rule moves the three-line paragraph to page 12.

If ICEF2 is presented with an oscillating document such as the one in this example, it will perform passes up to a limit that can be controlled by the user (the default limit is two passes). The system will then inform the user that the document failed to converge after the given number of passes and will report the line number of the reference tag responsible for the nonconvergence.

The ICEF2 data store

The process by which ICEF2 formats a document is shown in **Figure 3**. The user can interact with the source document by using an editor (this interaction is not shown in the figure). When the user issues the SHOW or PERFECT command, ICEF2 reloads its internal state from the appropriate stub and begins formatting. It takes lines from the source file, beginning at the point corresponding to the selected stub. GML tags found in the source text are interpreted by "tag routines" in an external "style definition." Some sample style definitions are provided with ICEF2, and additional styles may be defined by the user. During the process of formatting the document, a tag routine can invoke certain commands that save information in the ICEF2 data store for use by later passes, or recall information saved previously.

The ICEF2 data store is maintained by a small, special-purpose relational database system called the Data Manager. The design of the Data Manager is based on the following assumptions:

- The amount of data to be stored for each document is small.
- All data items to be stored are of character-string type.
- Ordering of data is important (e.g., the chapter headings in the table of contents should appear in the proper order).
- Retrieval of data is done in simple ways (e.g., by simple key-matching).
- The Data Manager should incorporate the notion of a "pass" through the document and provide special mechanisms to detect changes that occur from one pass to another.

Because the amount of data to be stored is small and fast access is desirable, the data store for a given document is kept in main memory during the processing of the document. At the end of the ICEF2 session, the data store is written out to a disk file keyed to the name of the document.

The data store is a collection of tables. Each table is an ordered list of tuples containing data values ("fields"). Each table has a name, but fields are not named. At any point in time, each table may have one of its tuples designated as "current" (or the "current" tuple for a table may be undefined).

The main-memory data structure for the ICEF2 data store is shown in **Figure 4**. The linked list of TABLE structures contains an entry for each table in the data store, including the name of the table, its number of fields, and a pointer to its first and current tuples. Each table has its own linked list of TUPLE structures that in turn contain pointers to the actual data values. The TABLE and TUPLE structures and the data values are all held in dynamic storage. Each tuple has three special flags (READ, WRITE, and CHANGE) that record whether the tuple has been read, written, or changed during the current pass. Each TABLE has a special flag called NOMORE that records the event that a search for a tuple in this table was unsuccessful during the current pass. Global static pointers called ANCHOR and LASTTABLE point to the first and last tables in the linked list, respectively.

In addition to the TABLE and TUPLE structures, the data store also has a static variable called WARNING, which serves as an indicator that one of the following events has occurred during a pass:

1. A search for a tuple is unsuccessful (this sets the NOMORE flag for the table involved), and a tuple is later inserted in that table.
2. A tuple is read and later changed.
3. At the end of a pass it is discovered that a tuple was read but not written.

Actions during a formatting pass

When a formatting pass begins, the overall WARNING flag for the data store, the NOMORE flag for each table, and the READ, WRITE, and CHANGE flags for each tuple in the data store are all turned off. During a pass, the tag routines that interpret the various tags encountered in the document may interact with the database using the following commands:

1. *DBFETCH* and *DBNEXT*: These commands are used to retrieve from the data store a tuple containing values that match certain key values. The tag routine provides the name of the table to be searched, indicates which fields are to be used as search keys, and provides a value for each key field. The key fields are not fixed properties of the table, but may be changed from one retrieval command to the next. DBFETCH searches the given table sequentially, beginning with the first tuple, returning the first tuple found that matches the given key values and making this tuple the "current" tuple for the table. DBNEXT behaves in exactly the same way as DBFETCH except that its search begins with the "current" tuple of the table rather than with the first tuple. If the tag routine specifies no key fields, DBFETCH unconditionally returns the first tuple, and DBNEXT unconditionally returns the "next" tuple and advances the current tuple pointer.

   When a tuple is retrieved by a DBFETCH or DBNEXT command, the data manager turns on its READ flag, indicating that the tuple was read during the current pass. If a DBFETCH or DBNEXT command fails to find a tuple matching the given keys, the data manager turns on the NOMORE flag for the given table, indicating that a search for data in that table was unsuccessful. If no table currently exists with the given name, an empty table is created with this name, and its NOMORE flag is turned on. The fact that the search was unsuccessful is indicated to the calling tag routine by a return code.

   An interesting special case occurs when a tag routine needs to retrieve *all* the tuples from a given table, as in fetching information on all the headings in the document to format a table of contents. Such a tag routine will make a call to DBFETCH with no key fields, followed by repeated calls to DBNEXT with no keys until a return code indicates that the table is exhausted. The last DBNEXT call always turns on the NOMORE flag for the table, since the retrieval attempt was unsuccessful (even though no search keys were provided). Any subsequent insertion of tuples into this table (caused, for example, by creation of a new heading in the document) will turn on the WARNING flag of the data store and force an additional formatting pass.
2. *DBSTORE*: This command is used by a tag routine to update information in an existing tuple of the data store or to create a new tuple. The calling tag routine names a table and provides a set of data values for a tuple of that table. The caller also indicates which of these values are to be considered "key fields." The DBSTORE command searches the given table for the first existing tuple whose stored values match the key fields provided. If such a tuple is found, its remaining fields are updated with the data values given by the DBSTORE command. If no tuple is found to match the given keys, a new tuple having the given keys and data values is inserted into the table. If no table currently exists with the given name, a new table is created with this name, and a tuple having the given keys and data values is inserted into it. It is important to note that key fields are not static properties of a table. A tuple may be inserted into a table by one key field and later retrieved by matching a different field.

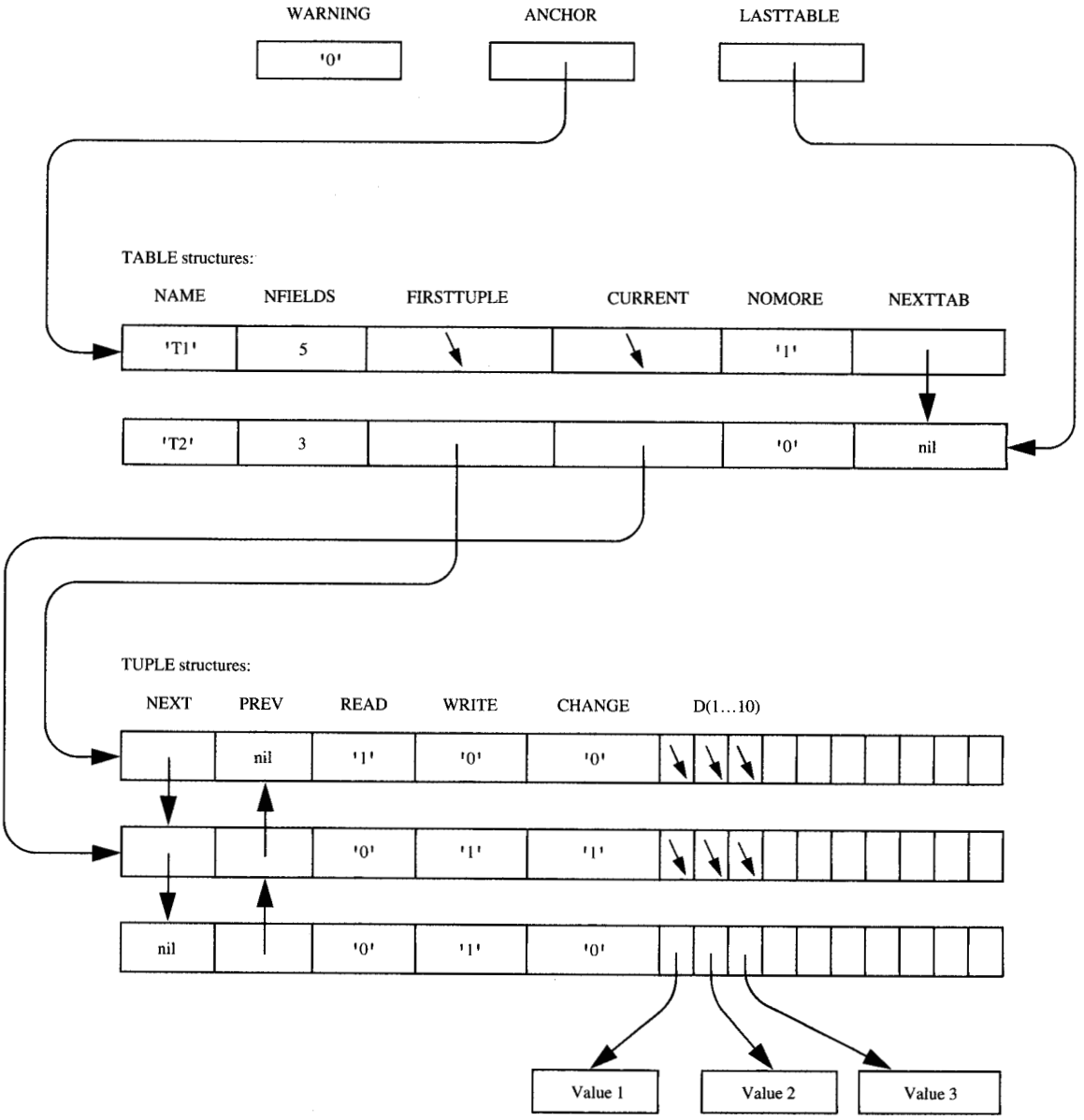


Figure 4

Structure of the ICEF2 data store in main memory.

The ordering among the tuples of a table is controlled by the values of their fields, taken from left to right, folded to uppercase (this folding is for sorting purposes only, and does not affect the stored values). All fields participate in the ordering, not just key fields. As the values of a tuple are updated, the tuple may change its position in the table. If a tuple is the "current" tuple of a table when its position changes, it remains "current."

DBSTORE always turns on the WRITE flag of the affected tuple, and, if the actual values of the tuple are changed, its CHANGE flag is turned on also. In addition, if the CHANGE flag is turned on and the READ flag was already on for this tuple, DBSTORE turns on the global WARNING flag of the data store. This records the fact that a data value was read and later updated in this pass through the document. The global WARNING flag is also

turned on if a new tuple is inserted into a table whose NOMORE flag is already on, indicating that data were added that might have affected the outcome of an earlier unsuccessful search.

Actions at the end of a pass

At the end of a formatting pass, flags indicate which tuples in the data store were read, written, and changed during the pass. In addition, the global WARNING flag for the data store is turned on if any tuple was read and later changed during the pass, or if data were inserted into a table that had previously been searched unsuccessfully.

The following sequence of events is triggered automatically at the end of each pass by the ICEF2 data store:

1. A "garbage-collection" scan is made of the entire data store, in which all tuples whose WRITE flags are turned off are deleted from the store. Since these tuples were not written during a complete pass through the document, they represent "dead" data which are no longer relevant. If any tuple deleted by this mechanism is found to have its READ flag turned on, the global WARNING flag is turned on, indicating that some "dead" data were retrieved by a tag routine during this pass.
2. After the "garbage-collection" step, the global WARNING flag is examined. If the WARNING flag is still turned off, the document has converged during the pass just completed, formatting is complete, and control is returned to the user. However, if the WARNING flag is turned on, the document has failed to converge during this pass. In this case, if the user-controlled limit on number of passes has been reached, control is returned to the user with a warning message. If the pass limit has not been reached, ICEF2 begins another formatting pass at the beginning of the document. Before beginning the next pass, it turns off the global WARNING flag, all the table NOMORE flags, and the READ, WRITE, and CHANGE flags for all the tuples.

At present, each pass initiated by ICEF2 begins on the first page of a document. In principle, however, a minor optimization is possible: It could be recorded in the "stub" of each page whether any data store references (DBSTORE, DBFETCH, or DBNEXT) were made during processing of the page. Each pass could then begin on the first page containing such a reference. Pages at the beginning of a document that do not in any way reference the data store will never change from one pass to the next and need not be reformatted.

Saving the data store in secondary storage

During an ICEF2 session, the data store is held in main memory for quick access. A more permanent version of the data store for each document is kept in secondary storage in order to record information about the document between formatting sessions. The content of the main-memory data store is saved in a persistent file whenever the user issues a SAVE or FILE command. The file format of the data store closely mirrors its main memory structure, except that the various flags are not saved since their values are not meaningful from one formatting session to another.

Example

This section describes a simple scenario that illustrates how the ICEF2 data store is used in resolving heading references and producing a table of contents. The scenario is somewhat oversimplified (for example, it ignores the facts that headings may occur in multiple levels and may be numbered).

Suppose that a document contains, among others, the following GML tags:

:toc.
Meaning: generate a table of contents.

(. . . various intervening materials . . .)

:hdref id='trains'.
Meaning: generate a reference to the heading whose id is 'trains'.

(. . . various intervening materials . . .)

:h1 id='trains'.Early Steam Locomotives
Meaning: generate a heading with an id of 'trains', containing the text *Early Steam Locomotives.*

Suppose further that the various tag routines that deal with headings have a convention of sharing information by means of a table named HEADS. Each heading is represented by one tuple in the HEADS table, which records its id, its text, and its page number. At the end of a previous formatting session, ICEF2 saved in secondary storage a HEADS table containing a tuple indicating that id='trains' corresponds to a heading on page 20 with text *Early Steam Locomotives.*

The following sequence of events might take place in a new ICEF2 session:

1. At the beginning of the session, the data store is loaded into main memory, its flags are reset, and the Safe Mark is set to the beginning of the document.
2. The user asks to see page 10 of the formatted document. The system begins formatting at the beginning of the document and proceeds toward the desired page. The :toc. tag is encountered first, and it reads the entire contents of the HEADS table and uses it to format the Table of Contents. By fetching all the tuples in the HEADS table and attempting to fetch more, the :toc. tag turns on the READ flags of all these tuples and also the NOMORE flag for the HEADS table.

3. The :hdref id='trains'. tag is encountered next. It fetches the *trains* tuple from the HEADS table by using id='trains' as a key, and uses the information to format a heading reference that looks like *"Early Steam Locomotives" on page 20.* The READ flag for the *trains* tuple would be turned on by this operation if it were not already on.
4. Page 10 is displayed to the user. While viewing this page the user inserts a new heading, using the following GML tag:

   :h1 id='boats'.Paddlewheel Steamboats

   The user requests that page 10 be redisplayed, which causes the new heading tag to be processed, and a new tuple is inserted into the HEADS table, indicating that the id *boats* refers to a heading with text *Paddlewheel Steamboats* on page 10. Insertion of this new tuple into HEADS causes the global WARNING flag of the data store to be turned on, because an insertion has been made into a table whose NOMORE flag was on.
5. The user asks to see page 25 of the formatted document. The system resumes formatting on page 10 and proceeds forward; on page 22 it encounters the tag :h1 id='trains'.Early Steam Locomotives. While processing this tag, the system updates the HEADS tuple with id='trains' in the data store, changing its page number from 20 to 22. The WRITE and CHANGE flags for this tuple are turned on. Since the READ flag was already on for this tuple, the global WARNING flag for the data store would be turned on by this operation if it were not on already. Formatting continues to page 25, which is displayed.
6. The user issues the PERFECT command, causing the system to continue formatting the document until it converges. The system formats from page 25 to the end of the document. Since the global WARNING flag is turned on, an additional pass is needed. All data store flags are turned off before the second pass begins.
7. On the second pass, when the :toc. tag is encountered, the system reads all the tuples from the HEADS table and generates a table of contents listing the *boats* heading on page 10 and the *trains* heading on page 22. The tag :hdref id='trains'. causes the *trains* tuple to be fetched by its id, and generates the reference *"Early Steam Locomotives" on page 22.*
8. As the :h1. tags are encountered for the two headings, they turn on the WRITE flags for their respective tuples, but not the CHANGE flags because the values stored in the tuples have not changed.
9. At the end of the second pass, all the tuples in HEADS have their WRITE flags on and their CHANGE flags off; the global WARNING flag is off, and the document has converged.

Storing page numbers

One of the most frequent uses of the ICEF2 data store is by tag routines wishing to record the page number on which some item occurs. By convention, if a tag routine calls DBSTORE to store a tuple in which some field has the value *&&&*, that value will be replaced by the current page number when the tuple is stored. When the value is subsequently retrieved by DBFETCH or DBNEXT, it will be a character string containing a page number such as *37* or *iv*. This convention is made possible by cooperation between two ICEF2 components called the Formatter and the Packer, described below.

An interesting special case occurs when a tag routine wishes to discover the page number on which its *own* contents will be placed. For example, a "figure reference" tag needs to know its own page number as well as the page number of the figure it refers to, since processing of the figure reference depends on whether these pages are the same or different. In this sense, the figure reference tag has two antecedents: itself, and the figure tag with a matching id attribute.

At the time when a tag routine is executing, the page on which its contents will be placed is not yet known. This is because of a feature of the ICEF2 architecture illustrated by **Figure 5**, a more detailed version of Figure 3. An ICEF2 style definition consists of three independent parts: the Syntax, the Tag Routines, and the Page Templates. These three parts of the style definition furnish instructions to three internal ICEF2 components called the Parser, the Formatter, and the Packer. The Syntax is simply a list of the valid tags in the document and their nesting rules, and it enables the Parser to recognize the tags and find their scopes. The Tag Routines are small Pascal procedures that process the individual tags, calling the Formatter to perform various functions such as justifying lines. The Tag Routines and the Formatter do not produce a complete formatted page, but instead generate a long column of justified text called the "galley." This galley is then sent to the Packer, which arranges the lines of text on pages according to rules in the Page Templates. This clean separation of the style definition into three independent parts has the advantage that each of the three parts is greatly simplified. For example, the tag routines do not contain any code for managing page properties such as margins or gutters, since all page properties are specified in the Page Templates.

The Formatter and the Packer communicate only by means of the galley. Since the galley is produced before pagination occurs, an individual tag routine can discover its own page number only by storing *&&&* in the data store and retrieving it on a subsequent pass. When a tag routine calls the Data Manager with a DBSTORE command to store a tuple containing *&&&*, the DBSTORE command is passed along to the Packer in the galley along with the current line of text. The Packer then replaces the *&&&* with the current

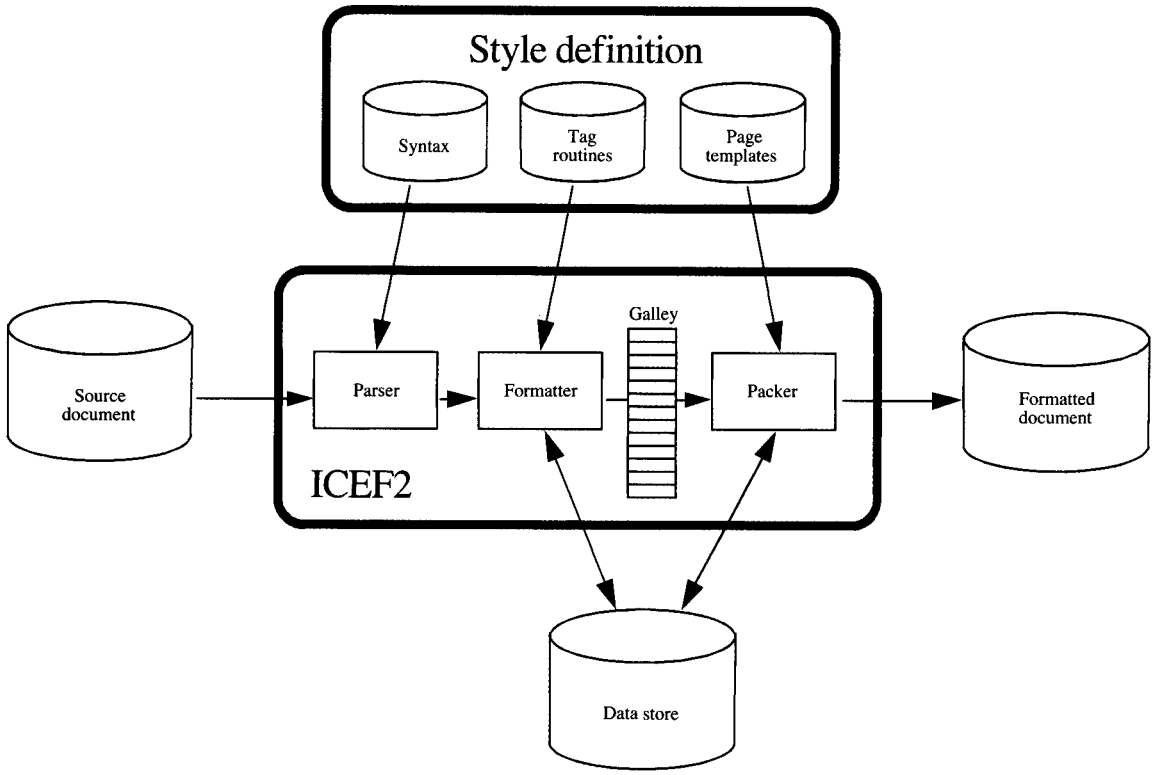


Figure 5

Details of the ICEF2 formatting process.

page number and executes the DBSTORE when this line of text is placed on a page.

On the first pass through a document, a tag routine that attempts to retrieve its own page number receives a return code from DBFETCH indicating "tuple not found," since the Packer has not yet stored the actual page number in the data store. By convention, the tag routine then substitutes a *?* for the missing page number. On subsequent passes, the tag routine can fetch the page number that was stored by the Packer on the previous pass. When the data store indicates that the document has converged, the page numbers fetched by the tag routines on the last pass are guaranteed to be valid.

We consider an example of a tag routine that needs to retrieve its own page number. Suppose we are trying to create a dictionary in which the first and last word defined on each page are printed at the top of the page. We write a portion of the style definition for the dictionary. Suppose that each definition in the source file for the dictionary is "marked up" by a :defn tag that takes the word to be defined as an attribute and contains the definition within its scope, as in the following example:

```
:defn word='cap'.An article of informal headgear.
:defn word='car'.A vehicle for transporting passengers.
:defn word='cat'.A furry four-legged animal.
```

The tag routine for the :defn tag must be written in such a way that the first and last definitions occurring on a page write their respective "word" attributes in a designated position at the top of the page.

ICEF2 provides a mechanism called a "fixture," which is a named area on a page into which a tag routine can place a string of characters. The name and location of the fixture are declared in the Page Template. A tag routine places text in the fixture by creating a named "box" of text and specifying "PLACEMENT=FIXTURE". The fixture-box is passed along to the Packer, which places it in the designated location on the current page. If several fixture-boxes with the same name are encountered during the packing of a page, the one to be

```
(* Global variables *)
var currentpage: string(5);  (* page number of current page *)
var firstword: string(20);   (* first word on current page *)

tag dict;
   (* Invoke this tag routine at the beginning of the document *)
   begin
      currentpage := '';     (* Initialize global variables to null  *)
      firstword := '';
      parse();               (* Process the contents of the document *)
   end;   (* end of DICT tag routine *)

tag defn;
   var pageno: string(5);    (* page number of this tag     *)
   var word: string(20);     (* word defined by this tag    *)
   var retcode: integer;     (* return code from data store *)

   begin

     (* obtain the word to be defined from the tag attribute *)
     word := argval('word');

     (* call the data store to find the page number of this word *)
     dbfetch2 (retcode, 'DEFTABLE', 'KD', word, pageno);
     if retcode ¬=0 then pageno := '0';  (* word not found *)
     (* pageno now contains the page number of this word,
        if known -- else zero *)

     (* If this is the first word on a new page, update FIRSTWORD *)
     if pageno ¬= currentpage then firstword := word;
     currentpage := pageno;

     (* Open a box and print the word in it *)
     command ('BOX HINGES=1L');        (* One line of space around box *)
     command ('FORMAT WEIGHT=BOLD');   (* Use bold face type           *)
     literal (word || '  ');           (* Put the word in the box      *)
     command ('FORMAT PREVIOUS');      (* Return to non-bold type      *)
     (* Note the word-box is still open *)

     (* Store the page number of this word in the data store *)
     dbstore2 (retcode, 'DEFTABLE', 'KD', word, '&&&');

     (* Create a fixture box, assuming this is the last word on the page *)
     command ('BOX NAME=TOPFIX PLACEMENT=FIXTURE WIDTH=6I');
     command ('FORMAT WEIGHT=BOLD');   (* Use bold face type      *)
     (* Write the first word and the current word in the fixture box *)
     literal (firstword || ' - ' || word);
     command ('FORMAT PREVIOUS');      (* Return to non-bold type *)
     command ('ENDBOX');               (* Close the fixture box   *)

     (* Now fill the word-box with the rest of the definition *)
     parse();
     command ('ENDBOX');               (* Close the word-box      *)
     (* The word-box may be packed partly on one page and partly on
        the next.  The fixture-box will take effect on the first
        page that contains any part of the word-box. *)

   end;  (* end of DEFN tag routine *)
```

Example 4

Tag routines used in formatting a dictionary.

printed is the last fixture-box encountered before the end of the page.

In our dictionary example, we define a fixture called TOPFIX to be printed at the top of each page. Each :defn tag routine updates the contents of the fixture-box. Each :defn tag routine assumes that its word is the last word to occur on the page; for the actual last word on the page, this assumption is true and the fixture-box is actually printed. Therefore, each :defn tag routine needs to write two words in the fixture-box: the first word on the current page, and its own word. The first word on the current page is stored in a global variable called FIRSTWORD. Each :defn tag routine fetches its own page number from the data store and compares it with the page number fetched by the previous :defn; if the page numbers are different, the current word is the first word on a page, and it is stored in FIRSTWORD and used by all subsequent occurrences of :defn until a new page number is encountered.

The actual tag routine for the :defn tag is shown in **Example 4**. Also shown in the same example is a fragment of a :dict tag routine, which must be invoked at the beginning of the dictionary to initialize certain global variables. A detailed explanation of the various ICEF2 commands invoked by the sample tag routines can be found in the *ICEF2 Installation and Style Definer's Guide* [17].

Because of the interactions between the Formatter and the Packer, formatting the dictionary based on these tags is a two-pass process. In the first pass, the page numbers associated with the various words are unknown. In the second pass, the tag routines are able to detect the first word on each page and format the fixture at the top of the page correctly.

Summary

This paper has described the mechanism of detecting document convergence in ICEF2, a formatter that permits users to reformat and display one page at a time. ICEF2 processes documents marked up with descriptive "tags" using the Generalized Markup Language. We have discussed in detail the handling of "reference" tags, the processing of which is dependent on information supplied by other tags. The formatting of a document containing such tags in general requires at least two passes, and in some cases results in an oscillating condition that can never be perfectly formatted.

ICEF2 permits tag-processing routines to share information by means of a simple relational database manager that has special features to detect when additional passes are needed to complete the formatting of a document. Information accumulated in the ICEF2 data store persists from one pass to the next and is saved in a file between ICEF2 sessions.

A case of particular interest to an ICEF2 style definer is a tag whose processing is dependent on its *own* page number. Because of the strong separation between justification (line-building) and pagination (page-building) in the ICEF2 system, the handling of such a tag becomes a two-pass process in which the current page number is communicated via the ICEF2 data store.

Limitations of the ICEF2 design include the facts that the smallest unit of reformatting is a page, and when an additional pass is called for, the entire document is reformatted. A promising area for further research is the design of a system that places much tighter bounds on the amount of processing required to achieve document convergence after an editing change. For example, if a change is made in a heading, it should be possible to find all the parts of the document (references, table of contents, etc.) that are impacted by the change and to reformat these parts locally without making a pass over the entire document.

Acknowledgments

The Interactive Composition and Editing Facility Version 2, IBM Program No. 5798-DTD, was developed at the IBM Almaden Research Center. In addition to the author, the following individuals were responsible for the design and implementation of this product: Olivier Bertrand, Jakob Gonczarowski, Michael J. Goodfellow, Daniel Mahoney, Mamata Misra, Dieter Paris, Carol H. Thompson, Stephen J. P. Todd, Bradford W. Wade, Daniel L. Weller, and Mitch Zolliker.

References and notes

1. R. Furuta, J. Scofield, and A. Shaw, "Document Formatting Systems: Survey, Concepts, and Issues," *Comput. Surv.* **14,** No. 3, 417–472 (1982).
2. *Document Composition Facility: Generalized Markup Language Starter Set Reference*, Order No. SH20-9187, 1985; available through IBM branch offices.
3. B. K. Reid, "Scribe: A Document Specification Language and Its Compiler," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, October 1980. Available as *Technical Report CMU-CS-81-100.* Scribe is a registered trademark of Unilogic Ltd., Pittsburgh, PA.
4. B. K. Reid and J. H. Walker, *Scribe Introductory User's Manual*, Second Edition, Carnegie-Mellon University, Pittsburgh, PA, 1979.
5. D. E. Knuth, *The TEXbook,* Addison-Wesley Publishing Co., Reading, MA, 1984. TEX is a trademark of the American Mathematical Society.
6. B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer* **16,** No. 8, 57–69 (1983).
7. C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs, "Alto: A Personal Computer," *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell, and A. Newell, Eds., McGraw-Hill Book Co., Inc., New York, 1982, pp. 549–572.
8. M. Hammer, R. Ilson, T. Anderson, E. Gilbert, M. Good, B. Niamir, L. Rosenstein, and S. Schoichet, "The Implementation of Etude, an Integrated and Interactive Document Production System," *SIGPLAN Notices* **16,** No. 6, 137–146 (1981).
9. MacWrite is a trademark of Apple Computer, Inc., Cupertino, CA.
10. J. Seybold, "The Xerox Professional Workstation," *The Seybold Report* **10,** No. 16, 16-3–16-18 (1981).

11. J. Seybold and D. Stivison, "Interactive Page Makeup," *The Seybold Report on Publishing Systems* **13,** No. 14, 14-3–14-25 (1984). Interleaf is a trademark of Interleaf, Inc., Cambridge, MA; Texet is a registered trademark of Texet, Inc., Cambridge, MA; Xyvision is a registered trademark of Xyvision, Inc., Woburn, MA.
12. J. Gutknecht, "Concepts of the Text Editor Lara," *Commun. ACM* **28,** No. 9, 942–960 (1985).
13. J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment," *Commun. ACM* **29,** No. 3, 184–201 (1986).
14. *Interactive Composition and Editing Facility Version 2: Availability Notice*, Order No. G320-0798, 1985; available through IBM branch offices.
15. *Interactive Composition and Editing Facility Version 2: User's Guide*, Order No. SH20-6724, 1985; available through IBM branch offices.
16. D. D. Chamberlin, O. P. Bertrand, M. J. Goodfellow, J. C. King, D. R. Slutz, S. J. P. Todd, and B. W. Wade, "JANUS: An Interactive Document Formatter Based on Declarative Tags," *IBM Syst. J.* **21,** No. 3, 250–271 (1982).
17. *Interactive Composition and Editing Facility Version 2: Installation and Style Definer's Guide*, Order No. SH20-6723, 1985; available through IBM branch offices.

Received November 6, 1985; accepted for publication July 7, 1986

Donald D. Chamberlin *IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120.* Dr. Chamberlin is a Research Staff Member in the Computer Science Department at Almaden Research Center. He joined IBM in 1971 at the T. J. Watson Research Center in Yorktown Heights. During his career with IBM he has worked on operating systems, relational database management systems, and document-formatting systems. He has received an IBM Outstanding Innovation Award for development of the SQL database language, and an Outstanding Technical Achievement Award for his work on the ICEF document-formatting system. Dr. Chamberlin received a B.S. degree from Harvey Mudd College, Claremont, California, in 1966, and a Ph.D. in electrical engineering from Stanford University, California, in 1971.